DEVELOPING GNU RADIO SIGNAL PROCESSING BLOCKS

André F. B. Selva (University of Campinas, Campinas, São Paulo, Brazil; andrefselva@gmail.com); André L. G. Reis (University of Campinas, Campinas, São Paulo, Brazil; andre.lgr@gmail.com); Karlo G. Lenzi (CPqD, Campinas, São Paulo, Brazil; lenzi@decom.fee.unicamp.br); Luis G. P. Meloni (University of Campinas, Campinas, São Paulo, Brazil; meloni@decom.fee.unicamp.br); Silvio E. Barbin (University of São Paulo, São Paulo, São Paulo, Brazil; barbin@usp.br)

ABSTRACT

GNU Radio is a popular toolkit for SDR development applications. Despite offering a large library of signal processing blocks, it is usually necessary to develop other blocks for new system requirements. Furthermore, complex applications, like Digital TV transceivers or LTE networks, for example, demand more complex digital signal processing blocks than the ones available in the GNU standard library. Thus, it is essential to know how to build custom blocks on GNU Radio if we desire to make full use of the GNU Radio features and Software Defined Radio platforms. This work aims to present an in-depth study on how to create new blocks in GNU Radio, clarifying dubious points generated from the lack of documentation available on the subject, which is based mainly on forums discussions and tutorials, and to serve as future reference for new developers. To better understanding this process and to cover all stages of development involved in the creation of blocks, we present a step-by-step procedure on how to build a byte interleaver, commonly used in wireless communication systems, such as WiMAX, LTE and DTV.

1. INTRODUCTION

GNU Radio is an open-source toolbox kit that provides a development environment and some processing blocks that can be used to create software defined radios. The software provides full integration with USRP boards beyond the device drivers specially designed for them [1].

GNU Radio applications are developed using the language Python, where the connection between signal processing blocks are made. A graphical user interface, GNU Radio Companion (GRC), is also provided. GRC could be used to simplify system design, just like Simulink does with projects developed with Matlab [2].

Despite GNU Radio Library provides more than a hundred signal processing blocks, the development of larger projects, like a Digital TV transmitter, or a LTE system, may probably require more complex operations.

In GNU Radio, the blocks are developed in C++. Despite there are already preliminary versions of Python-built blocks, this article will cover only the C++ approach.

Required knowledge to develop blocks would be some Object Oriented Programming concepts, signal processing techniques, familiarity with GNU/Linux operating system and good C++ programming skills (although some C familiarity could be enough to begin coding).

In the following sections we will present the block directory structure, the role of each file and library that we need to create with a block (and their structures), and finally develop a signal processing block fromscratch, as an example.

The examples shown here, as well as the shell scripts used on the compilation steps are tested on a Ubuntu 11.04, using GNU Radio 3.5.0.

2. CREATING A SIMPLE BLOCK

For the first step of this article, the basics of the block development will be discussed. The examples herein are based in [3]. For better understanding, we strongly recommend the download of the gr-my-basic file [4]. The file contains the example library we will study in depth from now on.

In this text and in the block's files, we follow the naming convention mainly adopted in GNU Radio. The name of a block is compounded by the library name, followed by its name, and a reference for its input and output data type. For example, a block from the library "gr_lib", named "add", that add two integer (i) inputs and converts the sum to float (f) will be named as gr_lib_add_if.

2.1. Library Example

Once the example has been downloaded and uncompressed, it should contain a series of directories and configuration

files. The creation process does not include the modification of all of them. Our work focuses on the following directories:

2.1.1 lib directory
This directory contains the source files for the block. Basically, it contains a header file (.h) and a C++ file (.cc) for each block in our library.

2.1.2 swig directory
SWIG (Simplified Wrapper and Interface Generator) is the tool that automatically generates a Python interface for the C++ blocks, so that they can be used in the Python flowgraphs. In the swig directory, there is a series of .i files, each one containing the data needed to create the Python interface. Additionally, the directory contains a .i file for the library.

2.1.3 grc directory
This directory contains the files necessary to create the GRC interface for the block. These files consist of .xml files describing the connections that the tool needs to do between GRC and the Python flowgraph.

2.1.3 Other directories
The other directories include a directory to examples and simple test applications (app), a directory containing a series of configuration files, which cannot be modified (config), and a directory for general python scripts (python).

2.2. Library Compilation

In this paper, we will cover the compilation method using autotools. Alternatively, cmake can be used, as indicated in [5]. Autotools are a group of programming tools GNU uses to make source code packages portable to different Unix-like systems and to reduce the amount of Makefiles that should be edited in the installation process. Due to the use of this tool, the library root folder contains some files, like AUTHORS, bootstrap, configure.ac, Makefile.am, Makefile.common, among others. A Makefile.am file is included on every folder of the library to provide Autotools with the configuration needed to compile the folder's content. Besides the block files themselves, this is the only extra code edition needed to make the library compilation by the addition of a new block.

```
1 $ sudo ./bootstrap
2 $ sudo ./configure
3 $ cd swig
4 $ sudo make generate-makefile-swig
5 $ cd ..
6 $ sudo make
7 $ sudo make install
8 $ sudo ldconfig
```

Fig. 1 – Library Compilation

The compilation process is quite simple. After opening a terminal window and changing the current directory to the library root folder, just execute the commands shown on Fig. 1.

The command executed on line 8 is only required at the first library compilation.

Now we will present in more details the three folders listed in this section and their corresponding files. Each folder contains some *Makefiles* that must be edited after the creation of each block. Only *Makefile.am* files need to be edited , and it is only necessary to add a reference for the new block's header and source file.

3. LIB DIRECTORY

As mentioned in the previous section, the lib directory is the one that contains all source files for the block which is a header file that contains the block's class declaration and definitions, and a C++ file that contains the definition of the block work.

In short, every block is created as a C++ class that inherits from a more comprehensive class, the gr_block class. Our job is to specify this new class, overwriting some of its methods and attributes, and, if needed, creating new ones.

Now, we will study the library downloaded as an example: gr_my. This library contains only one block, an amplifier that multiplies the amplitude of the input signal by a factor determined by the user. In the next subsections, we will take a closer look to each source file.

3.1. Header file

The header file (in the example gr_my_amplifier_ff.h) contains the class declaration. All GNU Radio blocks have their classes inheriting from a common class, the gr_block. The gr_block has three subclasses for simplification of three very usual kind of blocks: a) the gr_sync_block which is a block that consumes and produces an equal number of items per port, b) the gr_sync_interpolator where the number of input items is a fixed multiple of the number of output items (this constant factor between input and output must be determined in the source file) and c) the gr_sync_decimator, which is another fixed rate block, where the number of output items is a multiple of the number of input items predetermined by a constant.

In our example, the block is a synchronous block and so gr_sync_block will be used. After initial declarations, we declare the function gr_make_my_amplifier_ff (int k), whose type is an alias to the boost shared pointers, a type of

```
01  class gr_my_amplifier_ff : public gr_sync_block
02  {
03   private:
04   friend gr_my_amplifier_ff_sptr
          gr_make_my_amplifier_ff (int k);
05   int d_k;
06  gr_my_amplifier_ff (int k);  // private constructor

07   public:

08   int k () const { return d_k; }
09   void set_k (int k) { d_k = k; }
10  ~gr_my_amplifier_ff ();   // public destructor
11  int work (int noutput_items,
12            gr_vector_const_void_star &input_items,
13            gr_vector_void_star &output_items);

14   };
```

Fig. 2 – gr_my_amplifier_ff header file

pointers used to simplify storage issues. This function acts like the public interface for the block on other files (like swig files). More information about boost shared pointers can be obtained in [6].

So far, we have the class initialization. The private members of the class include a friend declaration of the make function, allowing gr_make_my_amplifier_ff to access the class private members, in special, the constructor. Besides, we have the declaration of a new attribute, d_k, which will store our gain information, and the declaration of the private constructor. A stretch of the header file is shown on Fig. 2.

The public members of the class are two methods that provide access to the private attribute, d_k, the destructor declaration, and the declaration of a function named work, where the entire job actually happens.

3.2. .cc file

The .cc file is where the signal processing routine is defined.

After the implementation of the make function, defined on the header file, there is the declaration of four constants: MIN_IN, MAX_IN, MIN_OUT and MAX_OUT, that indicate the minimum and maximum number of input and output streams the block will have.

Following this definition, the constructor's definition takes place. Here, we can see a gr_make_io_signature function. This function associates the block's input and output with the number of streams previously determined. Also, it indicates the data types of each stream. In this example, we do not need any code execution at the constructor. It is usual to use the constructor to allocate data structures we need to use on the signal processing. Further examples will cover this use.

Then, the destructor is overridden, but no code is required in this example.

Finally, we override the work function, where all the signal processing takes place. Let's take a deeper look at its code, shown on Fig. 3.

The parameters of the work function are the number of output items, a vector of pointers to input streams and a vector of pointers to output streams. In lines 4 and 5, we extract from these vectors a reference for the input and the output streams. After that, for every item produced (noutput_items), the output value is calculated: the input value multiplies by the gain parameter. For last, the function returns the number of output items, telling the runtime system how many items were produced.

4. SWIG DIRECTORY

SWIG , as already mentioned, is the tool that generates the interface between C++ blocks and Python. So, the library has some SWIG files (.i) associated, containing the information the tool will need to do its job.

The library contains a general file, which contains references to every block of the lib and it's corresponding header file and swig file. In the example, gr_my.i is the library swig file.

Besides the library information, each block has a .i file associated to its own. In the given example the file is gr_my_amplifier_ff.i. This file must contain shortened information about the block's class, reproducing the private and public class members we want to access from Python, and the declaration of the make constructor. Also, this file must have a call to GR_SWIG_BLOCK_MAGIC, which permits us to access the block as gr_my.amplifier_ff from Python.

5. GRC DIRECTORY

If we want to provide a graphical interface for the block, we

```
01  int gr_my_amplifier_ff::work (int noutput_items,
02  gr_vector_const_void_star &input_items,
03   gr_vector_void_star &output_items)
04   {
05  const float *in = (const float *) input_items[0];
06   float *out = (float *) output_items[0];

07   for (int i = 0; i < noutput_items; i++){
08     out[i] = in[i] * (float)d_k;
09   }

10  return noutput_items;
11}
```

Fig. 3 – Example of work function

need to create a .xml file specifying the correct interface for GNU Radio Companion.

The XML file contains a description of the visual appearance of the block on GRC, like input and output streams names, and some major structural information, like the parameters that should be passed to the code (like the gain, in the given example). It's also possible to propose a standard value for the parameter, or impose some restrictions on its value.

Further information about the XML interface is available on [7].

6. GR_BLOCK CLASS

After studying an introductory example, we are now able to take a deeper look on the gr_block class and its methods.

In the previous example, our block's class inherits from a subclass of gr_block: gr_synck block, already discussed in section 3.1. gr_block is a more comprehensive class, and the relation between the number of input and output items may not be constant or explicit. Instead of telling the run-time system the exact relation between input and output, we indicate, in the work function (general_work, actually, as we are working with gr_block class) the number of input items we consume from each input stream, using the *consume* function. The function needs as parameters the number of items consumed, and from which stream they come from. Alternatively, one may use consume_each function, which consumes the same amount of data from all input streams. In Fig. 4, we can see part of the implementation of the constant gain amplifier previously studied using gr_block class instead of gr_sync_block. One highlight is the use of general_work function, and the use of consume_each.

GNU Radio, during the execution of a Python flowgraph, bufferizes a certain amount of information and calls the general_work function. One very useful thing to know is how to control the quantity of data bufferized

```
01 int gr_my_amplifier_ff::general_work (int
02 noutput_items, gr_vector_int &ninput_items,
03 gr_vector_const_void_star &input_items,
04 gr_vector_void_star &output_items)
05 {
06  const float *in = (const float *) input_items[0];
07   float *out = (float *) output_items[0];
08
09   for (int i = 0; i < noutput_items; i++){
10     out[i] = in[i] * (float)d_k;
11 }
12   consume_each(noutput_items);
13   return noutput_items;
14 }
```

Fig. 4 - Example of the work function using gr_block

before the execution of general_work. To do so, we just have to override another gr_block method: forecast.

Forecast, by default, indicates a 1:1 relationship between input and output. In other words, the required number of input items will be the exact same number of output numbers to be produced. By overriding the forecast method we could implement a decimator or an interpolator by a way that not inheriting from gr_sync_decimator or gr_sync_interpolator, or we could require an exact amount of items to be buffered. This could be very useful when the input/output relation is not very simple, but could be determined with the processing of a fixed quantity of data. Fig. 5 illustrates the override of forecast.

Another important question is what to do if is required to preserve some information between the multiple bufferized executions of the signal processing routine. For example, let's suppose the block to be implemented is a randomizer, which uses a shifter register as a stage of the processing. Let's also suppose the data to be randomized is too large to be buffered, or it's even generated in real-time, making impossible to require a buffer large enough to contain all the information. If the multiple bufferized calls of the function are not considered by the programmer, every time one new call to the working function takes place, the shifter register will be cleared, leading to a periodic loss of information.

The simplest way to solve this problem is to remember that every block is a class, and we can define new attributes to it. So, in the randomizer example, one could declare a pointer to the shifter register's data type as an attribute of the randomizer's class. In the constructor, memory allocation could be placed, as correspondent memory deallocation could take place in the class' destructor. By this, our data will be preserved until the flowgraph stops its execution, eliminating the data loss problem.

7. AN ADVANCED EXAMPLE

In order to summarize most of the issues discussed in this paper, we will present a more complex signal processing

```
01   void gr_block:: forecast (int noutput_items,
02   gr_vector_int &ninput_items_required)
03   {
04    unsigned ninputs = ninput_items_required.size
();
05   for (unsigned i = 0; i < ninputs; i++)
06   // to create a 1:1 relation
07   // ninput_items_required[i] = noutput_items;
08   // to require 100 items on the input
09   ninput_items_required[i] = 100;
10)
```

Fig. 5 - Example of use of the forecast method

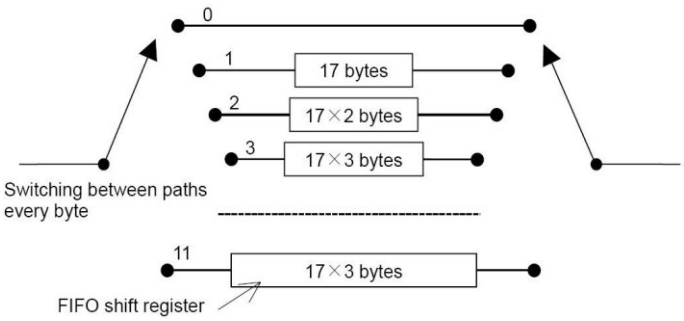

Fig. 6 - Byte interleaver

block example: a byte interleaver.

The byte interleaver here implemented is very common in digital communication systems, such as LTE and DTV standards [8]. The block's work is very simple: the input stream is redirected to different registers, from different sizes. Then, we recombine the shifters output, introducing some kind of shuffle on the bytes. This is very useful in the radio communication field, because it provides some data loss prevention in the transmission process. Fig. 6 illustrates the byte interleaving work.

The files herein referenced are available on [9].

7.1. lib/gr_my_byteinter_bb.h

First, let's take a look on the header file, in the lib folder. The block's class inherits from gr_block, so there is not a simple relationship between the input and output size. Besides similar structures we have already seen on section 3.1, the header files declares the public class members that can be seen on Fig. 7.

Each pointer to byte (v1 .. v11) indicates one path from the byte interleaver. The content of the shifters must be maintained between bufferized executions of the signal processing routine, so it is essential that this information is stored as attributes instead of as local variables. We also must preserve the information about the current shifter that must receive the next byte from the input stream. This information will be stored in the *state* variable.

```
1 public:
2
3 ~gr_my_byteinter_bb (); // public destructor
4  byte *v1, *v2, *v3, *v4, *v5, *v6, *v7, *v8, *v9,
5 *v10, *v11;
6
7  int state;
```

Fig. 7 - declaration of gr_my_byteinter_bb attributes

7.2. lib/gr_my_byteinter_bb.cc

After the declaration of the class, the .cc file is responsible to override the methods we need to use in the block.

First, we create a function called *shifter*, responsible to shift the data contained in each vector previously declared, returning the output number of the process, and receiving as parameters the size of the vector, the shift direction and the feedback input to the vector.

In the constructor, the shifter vectors are allocated, and properly initialized with zero. The *state* variable is also set to zero. In the destructor, all the vectors are freed from memory.

The general_work function makes use of the functions previously defined and implements the byte interleaving. As gr_byteinter_bb inherits from gr_block, we must indicate the number of items consumed from the input. Fig. 8 shows the general_work function code.

7.3. Other files

The example also requires the inclusion of a SWIG file and a XML file for GRC. It is also necessary to modify all Makefile.am files, and the SWIG file correspondent to the block's library.

```
01 int
02 gr_my_byteinter_bb:: general_work(int
03 noutput_items, gr_vector_int &ninput_items,
04 gr_vector_const_void_star &input_items,
05 gr_vector_void_star &output_items)
06 {
07  const byte *in = (const byte *) input_items[0];
08  const int ninput = (const int) ninput_items[0];
09  byte *out = (byte *) output_items[0];
10  for (int i = 0; i<noutput_items; i++){
11          out[i]=shifter(v1,state*M,'R',in[i]);
12          state++;
13          if (state == 12) state = 0;
14 }
15 this->consume(0,noutput_items);
16
17  // Tell runtime system how many output items we
18 produced.
19
20  return noutput_items;
21
22}
```

Fig. 8 – general_work function from gr_byteinter_bb.cc

8. FURTHER EXAMPLES

Readers willing to read further examples of block implementations are encouraged to visit GNU Radio's repository, where all natively included blocks' codes are available [10].

9. CONCLUSION

The development of a signal processing block with GNU Radio is very simple and allows the exploration of the full potential of this software. The examples illustrated here are very general and cover most part of user's common needs.

The final example, a byte interleaver, is a very common signal processing block which is part of many communication systems such as LTE, WiMAX and ISDB-T. Despite its real world application, the interleaver implementation is very simple.

10. REFERENCES

[1] Selva, A; Reis, A; Lenzi, K; Meloni, L; Barbin, S - A Software-Defined Radio Approach: GNU Radio. I2TS 2011 – Information and Telecommunication Technologies Conference Proceedings.

[2] Kaszuba, A. - MIMO Implementation with Alamouti Coding Using USRP2. Progress In Electromagnetics Research Symposium Proceedings 2011.

[3] How to program a new block for GNU Radio and GNU Radio Companion. http://gnuradio.org/redmine/attachments/download/271 – accessed on Apri 10, 2012.

[4] Library example used on this paper: http://gnuradio.org/redmine/attachments/download/276 – accessed on April 10, 2012.

[5] GNU Radio's official website Wiki page http://gnuradio.org/redmine/projects/gnuradio/wiki/CMakeWork – accessed on April, 10, 2012.

[6] Boost Smart pointers web page http://www.boost.org/libs/smart_ptr/smart_ptr.htm – accessed on April 10, 2012.

[7] GNU Radio's official website Wiki page – GRC section http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion#Creating-the-XML-Block-Definition – accessed on April, 10, 2012

[8] ABNT NBR 15601:2007, first edition – Digital terrestrial television – Transmission system. Brazilian Association of Technical Standards.

[9] http://www.rt-dsp.fee.unicamp.br/

[10] GNU Radio Repository - http://gnuradio.org/redmine/projects/gnuradio/repository/ - accessed on April10, 2012.